Lecture 21: Topics

- Huffman code and compression.
Lecture 22: Plan

- Ziv–Lempel text compression.
- Basic idea: Exploiting text repetitions.
- Features: Lossless compression; basis of zip, winzip, Tar, and numerous other commercial compression packages.
Compressing Text Beyond Huffman

A completely different approach was proposed by Yaacov Ziv and Abraham Lempel* in a seminal 1977 paper ("A Universal Algorithm for Sequential Data Compression", IEEE transactions on Information Theory).

Their algorithm went through several modifications and adjustments. The one used most these days is the modification by Terry Welch, in 1984, and known today as LZW compression.

Unlike Huffman, all variants of LZ compression do not assume any knowledge of character distribution. The algorithm finds redundancies in texts using a different strategy.

We will go through this important compression algorithm in detail.

*who later became professors at (correspondingly) the EE and CS faculties, Technion.
Huffman vs. Ziv Lempel: Basic Difference

Both Huffman and the codebook compressions are static. They compute frequencies, based on some standard corpus. These frequencies are used to build compression and decompression dictionaries, which are subsequently employed to compress and decompress any future text.

The statistics (or derived dictionaries) are either shared by both sides before communication starts, or have to be explicitly transmitted as part of the communication.

By way of contrast, Ziv-Lempel compression(s) are adaptive. There is no precomputed statistics. The basic redundancies employed here are repetitions, which are quite frequent in human generated texts.

There is no need here to share any data before transmission commences.
The basic idea of the Ziv-Lempel algorithm is to “take advantage” of repetitions in order to produce a shorter encoding of the text. Let $T$ be an $n$ character long text. In Python’s spirit, we will think of it as $T[0]T[1]\ldots T[n-1]$.

Suppose we have a $k$ long repetition ($k > 0$) at position $j$ and at position $p = j + m$ ($m > 0$), namely:


**Basic Idea:** Instead of coding $T[p]T[p+1]\ldots T[p+k-1]$ character by character, we can fully specify it by identifying the starting point of the first occurrence, $j$, and the length of the repetition, $k$. 
Ziv-Lempel: How to Represent Repetitions

Suppose we have a $k$ long repetition ($k > 0$) at positions $j, p$ ($j < p$): $T[j]T[j+1]...T[j+k-1]=T[p]T[p+1]...T[p+k-1]$.

There are two natural ways to represent the starting point, $j$. Either by $j$ itself, or as an offset from the second occurrence, namely $m$, where $m = p - j$.

The first option requires that we keep a full prefix of the text both while compressing and when decompressing. Since the text can be very long, this is not desirable. In addition, for long text, representing $j$ itself may take a large number of bits.
Suppose we have a \( k \) long repetition \((k > 0)\) at positions \( j, p \) \((j < p)\):

\[
\]

There are two natural ways to represent the starting point, \( j \). Either by \( j \) itself, or as an offset from the second occurrence, namely \( m \), where \( m = p - j \).

Instead of keeping all the text in memory, Ziv-Lempel advocates keeping only a bounded part of it. The standard recommendation is to keep the 4096-1 most recent characters.

This choice has the disadvantage that repeats below the horizon, \( i.e. \) earlier than 4096-1 most recent characters, will not be detected. It has the advantage that \( m \) can be represented succinctly \( 12 \) bits for 4096-1 size window).
High Level LZ Compression

To encode the string \( T[0]T[1] \ldots T[n-1] \), using a sliding window with \( W \) characters:

1. Loop over position in \( T \), starting with the index \( p=0 \)
2. While the text was not exhausted
   2.1 Find largest match for \( T[p:] \) starting at \( T[p-m:] \) for some \( 0 < m \leq W \).
   2.2 Suppose this match is of length \( k \), \( T[p-m:p-m+k] \), \( 2 \leq k \).
   2.3 Output \( m, k \).
3. Update the text location: \( p = p+k \).

Important: the overlapping segment \( T[p-m:p-m+k] \) may go beyond \( T[p] \) (into the “future”).

Remark: Recording repetitions of length 1 is wasteful in terms of bits used vs. bits saved. Thus the restriction \( 2 \leq k \).
We have already mentioned that the size of the window, $W$, is typically restricted to $4,096-1=4,095$. Thus the offset, $m$, can be represented by a fixed length, 12 bits number. The length of the match, $k$ is also limited, typically to 31. So the length, $k$, can be represented by a fixed length, 5 bits number.

Finding a maximum match quickly is also an important issue, determining the efficiency of the compression algorithm. Hashing and a trie data structure (to be discussed in the data structures course) are two possible approaches to speed up the computation. In both cases, we should be able to efficiently update the data structure before it becomes obsolete.
LZ Compression: Some More Details

We present a simple iterative procedure for the task, which does not employ any sophisticated data structures. Its performance (both in terms of running time, and of compression ratio) will not be as good as the optimized, commercial packages. But unlike the packages, you will fully understand what goes on here.

Text to ASCII (reminder):

```python
def str_to_ascii(text):
    """ Gets rid of non ascii characters in text""
    return ''.join(ch for ch in text if ord(ch)<128)
```
Maximum Match

Our first task is locating the maximum matches.

The function `maxmatch` returns the offset and the length of a maximum length match $T[p:p+k] == T[p-m:p-m+k]$ within prescribed window size backwards and maximum match size.

The function `maxmatch(T,p,w,max_length)` has four arguments:

- $T$, the text (a string).
- $p$, an index within the text.
- $w$, a size of window within which matches are sought.
- `max_length`, the maximal length of a match that is sought.

The last two arguments will have the default values $2^{12} - 1, 2^5 - 1$, respectively. With these default values, the offset can be encoded using 12 bits, and a match length can be encoded using 5 bits.
Maximum Match: Python Code

```python
def maxmatch(T, p, w=2**12-1, max_length=2**5-1):
    """ finds a maximum match of length k<=2**5-1 in a
    w long window, T[p:p+k] with T[p-m:p-m+k].
    Returns m (offset) and k (match length) """
    assert isinstance(T,str)
    n = len(T)
    maxmatch = 0
    offset = 0
    for m in range(1, min(p+1, w)):
        k = 0
        while k < min(n-p, max_length) and T[p-m+k] == T[p+k]:
            k += 1
            # at this point, T[p-m:p-m+k]==T[p:p+k]
        if maxmatch < k:
            maxmatch = k
            offset = m
    return offset, maxmatch
    # returned offset is smallest one (closest to p) among
    # all max matches (m starts at 1)
```
```
Maximum Match: Computational Complexity

For any location, \( p \), this function takes up to \( w \cdot \text{max}_\text{length} \) many operations in the worst case.

For the default parameters, this is \( 2^{12} \cdot 2^5 = 2^{17} \) per one position, \( p \).

This is a rather pessimistic worst case estimate, as it assumes that in every position, we go all the way to length 31 and do not find a mismatch earlier.

Running \( \text{maxmatch}(T, p) \) over all text locations will thus take up to \( 2^{17} \) times the length of \( T \) operations. This is the major consumer of time in our compression procedure.

We will later sketch (and only sketch) ideas for improving this bottleneck.
Maximum Match: A Few Experiments

```python
>>> s="aaabbbbaabbbbaaa"
>>> lst=[maxmatch(s,i) for i in range(len(s))]
>>> print(lst)
[(0, 0), (1, 2), (1, 1), (0, 0), (1, 2), (1, 1), (6, 9), (6, 8),
 (6, 7), (6, 6), (6, 5), (6, 4), (6, 3), (1, 2), (1, 1)]
```

```python
>>> s='how much wood would the wood chuck chuck if the wood chuck would chuck wood should could hood'
>>> lst=[maxmatch(s,i) for i in range(len(s))]
>>> print(lst)
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (7, 1),
 (5, 1), (7, 1), (9, 1), (1, 1), (0, 0), (5, 3), (5, 2), (4, 1),
 (11, 1), (0, 0), (6, 2), (6, 1), (0, 0), (14, 1), (0, 0), (15, 6),
 (15, 5), (15, 4), (15, 3), (9, 2), (5, 1), (23, 2), (9, 1),
 (26, 2), (3, 1), (0, 0), (6, 7), (6, 6), (6, 5), (6, 4), (6, 3),
 (6, 2), (6, 1), (0, 0), (0, 0), (24, 16), (24, 15), (24, 14),
 (24, 13), (24, 12), (24, 11), (24, 10), (24, 9), (24, 8),
 (18, 7), (18, 6), (18, 5), (18, 4), (18, 3), (18, 2), (45, 7),
 (45, 6), (45, 5), (45, 4), (45, 3), (12, 10), (12, 9), (12, 8),
 (12, 7), (12, 6), (12, 5), (12, 4), (23, 6), (23, 5), (23, 4),
 (23, 3), (11, 2), (5, 1), (0, 0), (77, 2), (18, 6), (18, 5),
 (18, 4), (18, 3), (18, 2), (15, 1), (6, 5), (6, 4), (6, 3),
 (6, 2), (6, 1), (12, 2), (18, 3), (18, 2), (5, 1)]
```
Our Version of LZW Compression

Instead of producing a string consisting of bits right away, we decompose this task to two: An intermediate output (in an intermediate format), and then a final output. The intermediate format will be easier to understand (and to debug, if needed).

At the first stage, we produce a list, whose elements are either single characters (in case of a repeat of length smaller than 2), or pairs $[m, k]$, where $m$ is an offset, and $k$ is a match length. The default bounds on these numbers are $1 \leq m < 2^{12}$ (12 bits to describe) and $2 \leq k < 2^5$ (5 bits to describe).
At the first stage, we produce a list, whose elements are either single characters (in case of a repeat of length smaller than 2), or pairs \([m, k]\), where \(m\) is an offset, and \(k\) is a match length. The default bounds on these numbers are \(1 \leq m < 2^{12}\) (12 bits to describe) and \(2 \leq k < 2^5\) (5 bits to describe).

The algorithm scans the input text, character by character. At each position, \(p\), it invokes \(\text{maxmatch}(\text{text}, p)\). If the returned match value, \(k\), is 0 or 1, the current character, \(\text{text}[p]\), is appended to the list. Otherwise, the pair \([m, k]\) is appended.

If a match \([m, k]\) was identified, we advance the location in the text to be examined next from the current \(p\) to \(p+k\) (why?).
def LZW_compress(text, w=2**12-1, max_length=2**5-1):
    """LZW compression of an ascii text. Produces a list comprising of either ascii characters or pairs [m,k] where m is an offset and k is a match (both are non negative integers) """
    result = []
    n = len(text)
    p = 0
    while p<n:
        m,k = maxmatch(text, p, w, max_length)
        if k<2:
            result.append(text[p])  # a single char
            p += 1
        else:
            result.append([m,k])  # two or more chars in match
            p += k
    return result  # produces a list composed of chars and pairs
Of course, compression with no decompression is of little use.

```python
def LZW_decompress(compressed, w=2**12-1, max_length=2**5-1):
    """LZW decompression from intermediate format to ascii text""
    result = []
    n = len(compressed)
    p = 0
    while p<n:
        if type(compressed[p]) == str:  # char, as opposed to a pair
            result.append(compressed[p])
            p+=1
        else:
            m,k = compressed[p]
            p += 1
            for i in range(0,k):
                # append k times to result;
                result.append(result[-m])
                # fixed offset m "to the left", as result itself grows
    return """.join(result)
```
Intermediate Format LZW Compression and DeCompression: A Small Example

```python
>>> s="abc"*20
>>> 7*len(s)
420
>>> inter= LZW_compress(s)
>>> s
'abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc'
>>> inter
['a', 'b', 'c', [3, 31], [3, 26]]
>>> binn = inter_to_bin(inter)
>>> len(binn)
60

>>> inter2 = bin_to_inter(binn)
>>> inter2==inter
True
```

As demonstrated above, compression could pass the current location: That is, we can have \( j+k-1 > p \) in
Intermediate Format LZW Compression and DeCompression: Another Small Example

```python
>>> s="""how much wood would the wood chuck chuck if the wood chuck would chuck wood should could hood"""

```
Let us complete our LZW tour by going from the intermediate format to the compressed string of bits, and vice versa. We expect not to run into either Golumn nor Sauron along the way. Indeed, the code for both transformations is simple and efficient.
To distinguish between a single character and an \([m,k]\) entry, a '0' will be placed before a single ascii character, while a '1' will be placed before an \([m,k]\) entry.

```python
def inter_to_bin(lst, w=2**12-1, max_length=2**5-1):
    """ converts intermediate format compressed list to a string of bits""
    offset_width = math.floor(math.log(w,2)) + 1
    match_width = math.floor(math.log(max_length,2)) + 1
    #print(offset_width,match_width) # for debugging
    result = []
    for elem in lst:
        if type(elem) == str:
            result.append("0")
            result.append( (bin(ord(elem))[2:]).zfill(7) )
        elif type(elem) == list:
            result.append("1")
            m,k = elem
            result.append( (bin(m)[2:]).zfill(offset_width) )
            result.append( (bin(k)[2:]).zfill(match_width) )
    return "".join(ch for ch in result)
```

Don’t forget to \textbf{import} \texttt{math} for the logarithm.
def bin_to_inter(compressed, w=2**12-1, max_length=2**5-1):
    """ converts a compressed string of bits
    to intermediate compressed format """
    offset_width = math.floor(math.log(w,2)) + 1
    match_width = math.floor(math.log(max_length,2)) + 1
    #print(offset_width,match_width)  # for debugging
    result = []
    n = len(compressed)
p = 0
    while p<n:
        if compressed[p] == "0":  # single ascii char
            p += 1
            char = chr(int(compressed[p:p+7], 2))
            result.append(char)
p += 7
        elif compressed[p] == "1":  # repeat of length > 2
            p += 1
            m = int(compressed[p:p+offset_width],2)
p += offset_width
            k = int(compressed[p:p+match_width],2)
p += match_width
        result.append([m,k])
    return result

Don’t forget to import math for the logarithm.
There and Back Again: The Compress/Decompress Cycle

Don’t forget to import math for the logarithm.

```python
>>> text = """how much wood would the wood chuck chuck if the wood chuck would chuck wood should could hood""
>>> inter = LZW_compress2(text)
>>> comp = inter_to_bin(inter)
>>> comp
'0110100001101111011101110010000011011010111010101100011011010001000001110111011011100011101110110111011011101110101100100100000000011101101111011010001001000001101000011001011000000011100110011001101001011001101000000011000111011010010110011010000000110001000010000001011010011010000000011000111011010010100101101100011001000010000001110100011010000110010110000000011110011001100011011010001000000100100011010000001100010001010000001011001100011000110100110000001001000110100000011000100000001011100011011100110110100010000001001000110100000000110001110110100101100110100000001100010001000000101101001101000000001100010001000000111010001101110101011011000011101000110100001100101100000000111100110011000110110100010000001001000110100000000110001010110100010000001001000011000000

>> inter2 = bin_to_inter(comp)
>>> inter==inter2
True
>>> LZW_decompress(inter2)
'how much wood would the wood chuck chuck if the wood chuck would chuck wood should could hood'

Does it convince you the code is fine? As a toy example, it is not bad. But I would strongly recommend more extensive testing with substantially longer text, going through a larger number of cases.
For convenience, we “package” together all relevant functions.

```python
def process1(text):
    """ packages the whole process using LZ_compress """
    atext = str_to_ascii(text)
    inter = LZW_compress(atext)
    binn = inter_to_bin(inter)
    inter2 = bin_to_inter(binn)
    text2 = LZW_decompress(inter)
    return inter, binn, inter2, text2
```
There and Back Again: The NY Times Test

```python
>>> text = urllib.request.urlopen('http://www.nytimes.com/').read()
>>> clean_ny_text = str_to_ascii(text.decode('utf-8'))
>>> inter, binn, inter2, text2 = process1(clean_ny_text)
>>> print(text2 == clean_ny_text) # should be True
True

Now I am ready to believe the code is OK (of course this is by no means a proof of correctness).

```python
>>> (len(clean_ny_text)*7, len(binn))
(896105, 329290)
>>> 329290/896105
0.3674680980465459 # 37% of original
```

It seems that human generated text is more amenable to Ziv-Lempel compression than to Huffman compression.
There and Back Again: Some NY Times Text

```python
>>> print(clean_ny_text[310:480])
Find breaking news, multimedia, reviews & opinion on Washington, business, sports, movies, travel, books, jobs, education, real estate, cars & more."
<meta name=

>>> print(nytext2[30191:30475])
Eastman Kodak Files for Bankruptcy Protection</a></h5>
<h6 class="byline">
By MICHAEL J. DE LA MERCED
<span class="timestamp">1 minute ago</span>
</h6>
<p class="summary">
The 131-year-old film pioneer said on Thursday it would continue operating normally during bankruptcy.

Lots of html code and other “contents supporting” text, evidently.
Lots of html code and other “contents supporting” text, evidently.

The following piece of code is helpful in cleaning well formatted html tags:

```python
import re  # good old regexp package

def cleanhtml(raw_html):
    cleanr = re.compile(' <.*?> ')
    cleantext = re.sub(cleanr, '', raw_html)
    return cleantext
```

Incidentally, this is the 19 Jan 2012 NY Times issue. But we’ll try today’s issue in class. We expect around 250,000 character long text, and running our slow implementation of LZW may be too slow for an online show. But we will give it a try!
The function `maxmatch(text, p, w, max_length)` is the major consumer of time. It is invoked for every location `p` that was not skipped over. For locations far enough from the boundary, and for the default parameters, this is $2^{12} \cdot 2^5 = 2^{17}$ operations per invocation in the worst case.

Let us assume that one half of the text is skipped over (a reasonable assumption for human generated text). If `n` denotes the text length, compression will require $2^{16} \cdot n$ operations.

We will now compress a moderately long string, the proteome of the cholera bacteria, and examine the time it would take our code to compress it.
Measuring Time for Compression

```python
>>> cholera = open("Vibrio_cholerae_B33.txt").read()
>>> cholera_process = process1(cholera)
>>> print(cholera_process[3] == cholera)
True  # sanity check

>>> t1 = time.time()
>>> cholera_process = process1(cholera)
>>> t2 = time.time()
>>> print(t2-t1, (t2-t1)/60)
2647.4715678691864 44.124526131153104 # over 44 minutes

>>> print(len(cholera_process[1])/(7*len(cholera)),"\n")
0.8749265633656836  # compression ratio
```

So, on an iMac, 2014 desktop, 3.5 GHz Intel Core i5 processor, compression took 44 minutes. The compression ratio is 87.5% – not as good as for human generated text.
We saw that the Ziv-Lempel algorithm compresses the Cholera proteome to 87.5% of its original size. The Cholera proteome is (to the best of our knowledge) not human made. So some properties common in human generated text, like repetitions, are not too frequent. Thus the Ziv-Lempel compression ratio is not very impressive here. However, most of the cholera proteome text is over the amino acid alphabet, which has just 20 characters. The vast majority of the characters in the text can thus be encoded using under 5 bits on average. This indicates that maybe Huffman could do better here.
def process_cholera():

    cholera = open("Vibrio_cholerae_B33.txt").read()
    print("cholera length in bits", len(cholera)*7)
    cholera_count = char_count(cholera)
    cholera_list = build_huffman_tree(cholera_count)
    cholera_encode_dict = generate_code(cholera_list)
    cholera_decode_dict = reverse_dict(cholera_encode_dict)
    cholera_compressed = compress(cholera, cholera_encode_dict)
    print("compressed choleratext length in bits",
          len(cholera_encoded_text))
    print("compression ratio",
          len(cholera_compressed)/(len(cholera)*7))
    cholera_decoded_text = decompress(cholera_encoded_text,
                                      cholera_decode_dict)
    return cholera, cholera_decoded_text,
            cholera_encode_dict, cholera_decode_dict
Cholera Compression by Huffman: Execution

```python
>>> cholera_text, cholera_decoded_text,
    cholera_encode_dict, cholera_decode_dict = process_cholera()

cholera length in bits 21281953
compressed cholera text length in bits 15650235
compression ratio 0.7353758839707991
    # much better than LZW 0.874 ratio

```n

```python
>>> cholera_text == cholera_decoded_text
True    # sanity check

```n

```python
>>> count = char_count(cholera)

```n

```python
>>> scount = sorted(count.items(), key = lambda x:x[1])

```n

```python
>>> scount[-10:]    # 10 most popular chars
[(‘D’, 119431), (‘Q’, 121244), (‘T’, 122391), (‘I’, 141045),
 (‘E’, 147650), (‘S’, 148305), (‘G’, 156634), (‘V’, 172055),
 (‘A’, 217096), (‘L’, 252522)]

```n

```python
>>> cdict = cholera_encode_dict

```n

```python
>>> sdict = sorted(cdict.items(), key = lambda x:len(x[1]))

```n

```python
>>> sdict[:10]     # 10 shortest encoding
 (‘G’, ’0101’), (‘I’, ’0001’), (‘S’, ’0011’), (‘D’, ’11010’),
 (‘F’, ’10001’), (‘N’, ’01111’)]

```n
Intermediate Format LZW Compression: A Small Improvement

To encode a repetition using the default parameters, we use one bit to indicate this is a pair, 12 bits for $m$, and 5 bits for $k$. All by all, such encoding is $1+12+5=18$ bits long for all $k$, $k < 2^5$. A “single ascii character” entry is 8 bits long, so two single characters take 16 bits.

We conclude that encoding a two character repeat takes 18 bits, which is always longer than encoding the two separately. Furthermore, once in a while the second character will be the beginning of a different, longer repeat.

So it makes sense to encode only repetitions of length 3 or more. We note that the existing decompression code works just fine for both the “older” and the “improved” versions of compression.
def LZW_compress2(text, w=2**12 -1, max_length=2**5 -1):
    """LZW compression of an ascii text. Produces a list comprising of either ascii characters or pairs [m,k] where m is an offset and k>3 is a match (both are non negative integers) """
    result = []
    n = len(text)
    p = 0
    while p<n:
        m,k = maxmatch(text, p, w, max_length)
        if k<3: # modified from k<2
            result.append(text[p]) # a single char
            p += 1 #even if k was 2 (why?)
        else:
            result.append([m,k]) # two or more chars in match
            p += k
    return result # produces a list composed of chars and pairs
Testing the 2 to 3 Improvement

We start by packaging (almost identical to \texttt{process1}), and then run it on a \textit{toy example}.

\begin{verbatim}
def process2(text):
    """ packages the whole process using LZ_compress2 """
    atext = str_to_ascii(text)
    inter = LZW_compress2(atext)
    binn = inter_to_bin(inter)
    inter2 = bin_to_inter(binn)
    text2 = LZW_decompress(inter)
    return inter, binn, inter2, text2

>>> a = process1("aabaacaad")
>>> b = process2("aabaacaad")
>>> a[0]  # intermediate
[‘a’, ‘a’, ‘b’, [3, 2], ‘c’, [3, 2], ‘d’]
>>> b[0]  # intermediate
>>> len(a[1])  # binary
76
>>> len(b[1])  # binary
72
\end{verbatim}
We now run `process2` on the cholera proteome, check its performance (run time and compression ratio) and compare them to `process1` and to Huffman.

```python
>>> t0 = time.time()
>>> cholera_process = process2(cholera)
>>> t1 = time.time()
>>> print(cholera_process[3]==cholera) # sanity check
True
>>> print(t1-t0,(t1-t0)/60)
2750.9522049427032 45.84920341571172 #45 min, comparable to process1
>>> print(len(cholera_process[1])/(7*len(cholera)))
0.7896125886566896
```

So, on the same iMac (late 2014 desktop, 3.5 GHz Intel Core i5 processor), improved `process2` took 45 minutes. The compression ratio is 78.9% – not as good as Huffman’s 73.5% but better than `process1` 87.5%.
Improvements to LZW: gzip

The gzip variant of LZW was created and distributed (in 1993) by the Gnu† Free Software Foundation. It contains a number of improvements that make compression more efficient time wise, and also achieves a higher compression ratio.

As we saw, finding the offset, match pairs \([m,k]\) is the main computational bottleneck in the algorithm. To speed it up, gzip hashes triplets of consecutive characters. When we encounter a new location, \(p\), we look up the entry in the hash table with the three character key \(T[p]T[p+1]T[p+2]\). The value of this key is a set of earlier indices with the same key. We use only these (typically very few) indices to try and extend the match.

\[\text{†The name “GNU” is a recursive acronym for “GNU’s Not Unix!”; it is pronounced g-noo, as one syllable with no vowel sound between the g and the n.}\]
To prevent the hash tables from growing too much, the text is chopped to blocks, typically of 64,000 characters. Each block is treated separately, and we initialize the hash table for each. Hashing improves the running time substantially. To improve compression, gzip further employs Huffman code\(^\dagger\). This is used both for the characters and for the offsets (typically close offsets are more frequent than far away ones) and the match lengths.

For every block, the decoding algorithm computes the corresponding Huffman code for all three components (characters, offsets, matches). This code is not known at the receiving end, so the small table describing it is sent as part of the compressed text.

\(^\dagger\)such combination is sometime termed the Deflate compression algorithm.
Compression: Concluding Remarks

There are additional variants of text compression/decompression algorithms, many of which use combinations of Ziv-Lempel and Huffman encoding. In many cases, it is possible to attain higher compression by employing larger blocks or longer windows.

Our compression algorithm as described so far is greedy: Any repeat of length 3 or more is reported and employed right away. Sometimes this is not optimal: We could have an \([m_1, k_1]\) repeat in position \(p\), and an \([m_2, k_2]\) repeat in position \(p+1\) or \(p+2\), with \(k_1 \ll k_2\). Thus a non-greedy algorithm may result in improved compression.

All such improvements would cost more time but produce better compression. In some applications, such tradeoff is well justified.

Compression of gray scale and color images, as well as of documents with a mixture of images and text, uses different approaches. These are based on signal processing techniques, which are often lossy, and are out of scope for our course.